

PP lab PrOJECT: Parallelization of BFS

**CSE-A**

**GROUP MEMBERS:**

**Pallabi Chandra (160905011) roll no: 11**

**Abirami Byju(160905097) roll no: 31**

**Nikila Pulakhandam(160905099) roll no: 35**

**ABSTRACT**

Breadth-First Search (or BFS) is an algorithm for searching a tree or a graph data structure. Here, we start with a node and then visit all the adjacent nodes in the same level and then move to the adjacent successor node in the next level. This is also known as **level-by-level search**.

This project focuses on implementation of BFS graph using CUDA (Compute Unified Device Architecture) and MPI. Large graphs are common in various engineering fields and it is important for them to have faster execution which is why parallel computation is necessary. **Parallel algorithm** is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.

**OBJECTIVE**

Some of the objectives of this project is to show the reduced time for processing many vertices from a large graph and also to show GPU’s are better to use rather than normal CPU’s to process large data by using best appropriate algorithm.

This project searches an unweighted graph and returns order in which vertices are visited using CUDA and MPI.

**INTRODUCTION**

The BFS algorithm is an uninformed search technique that follows the following steps-

1. Start with the root node, mark it visited.
2. As the root node has no node in the same level, go to the next level.
3. Visit all adjacent nodes and mark them visited.
4. Go to the next level and visit all the unvisited adjacent nodes.
5. Continue this process until all the nodes are visited

The applications of BFS are-

**Shortest Path and Minimum Spanning Tree for unweighted graph:**In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

**Peer to Peer Networks:** In Peer to Peer Networks like [BitTorrent](https://www.geeksforgeeks.org/how-bittorrent-works/), Breadth First Search is used to find all neighbor nodes.

**Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.

**Social Networking Websites:**In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.

**LITERATURE REVIEW**

1. <https://www.ijcat.org/IJCAT-2014/1-3/Parallelization-of-BFS-Graph-Algorithm-using-CUDA.pdf>

This paper contains the basic objective and idea for the project.

2.<https://www.nvidia.co.uk/content/cudazone/CUDABrowser/downloads/Accelerate_Large_Graph_Algorithms/HiPC.pdf>

This paper contains the algorithm used in CUDA as well as the sample graph that we have used to demonstrate the codes working.

**METHODOLOGY**

**MPI Code:**

1.We use an adjacency matrix in order to take input for a directed graph. We use Collective communication routines in order to scatter the values of the adjacency matrix between a number of processes (level wise).

2. Initially the adjacency queue for each node is initialized to -1. As each process traverses through the values of adjacency list, if it finds an edge to the node, it places the appropriate node in the queue.

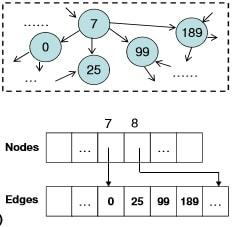
3.The values of adjacency queue are then gathered. The allvisited() function checks if BFS is complete or not. If it is, then the BFS traversal order is printed.

**CUDA Code:**

In this approach we should  traverse from a selected node (source or starting node) and traverse the graph in the form of layers implying we need to keep exploring the neighboring nodes to the currently processed node. A direct implementation of the traditional BFS algorithm on the GPU may not be possible, since CUDA has a restrictive programming model and is tricky to use as well.

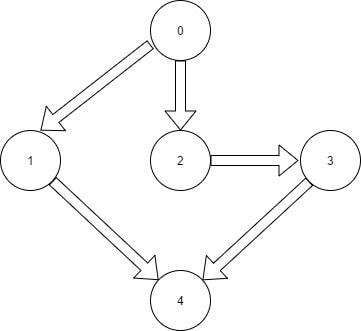
Instead of the traditional implementation of the BFS algorithm using a queue and Adjacency Matrix/List as graph representations, we’ve implemented a variation of the BFS Algorithm as proposed by P Harish and PJ Narayan in their paper [Accelerating large graph algorithms on the GPU using CUDA](http://www.nvidia.co.uk/content/cudazone/CUDABrowser/downloads/Accelerate_Large_Graph_Algorithms/HiPC.pdf).

In their paper, they present a different data structure to represent graphs in what is called a compact adjacency list form. In this structure, vertices of the graph are present in the array  Va. Another array  Ea of adjacency lists stores the edge vertices for all the vertices in the graph, such that each entry in the Vertex Array  Va refers to the starting index of its adjacent neighboring vertices in  Ea (Each entry of Ea refers to a vertex in the vertex array Va ). Refer to the below picture for clarity:

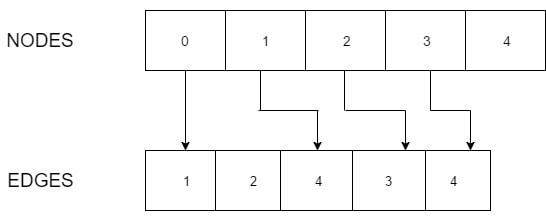


In the example the nodes adjacent to 7 are obtained from E[ V[7] ] to E[ V[8] - 1].

I’ll test my implementation on the below graph for the time being:

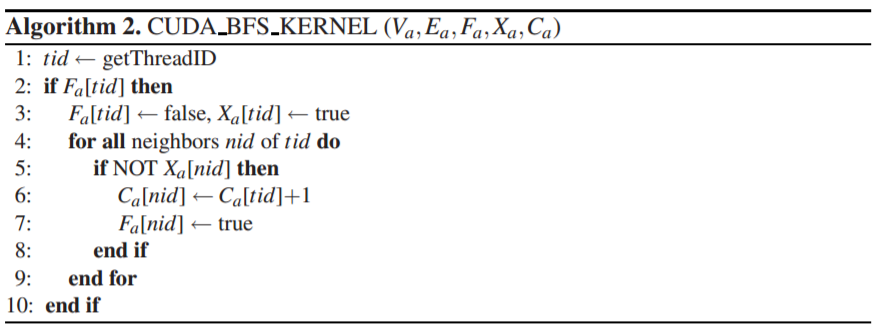


The corresponding representation of the above graph is as follows:

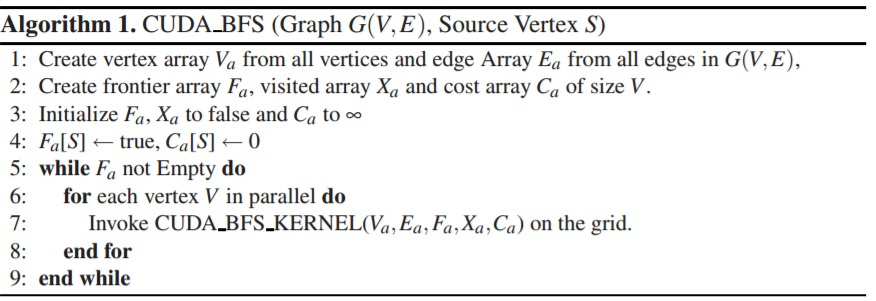


The paper attempts to solve BFS using level synchronization. They choose not to maintain a queue for each vertex during BFS since a queue would incur additional overheads of maintaining array indices and changing configuration. They instead maintain a frontier array of vertices which represent the vertices all the same level that are currently being processed. We’ve represented this array using boolean type. Another boolean array is the visited array, which has the same implications as the similarly named array in the traditional BFS algorithm. They also have a cost array which stores the minimal number of edges for every vertex from the source vertex.

Alhgorithm-

Kernel: 

Host code:



  We’ve used a structure to represent every vertex, which is identified as an index in the vertex array and contains two properties:

* start: Index of the first adjacent node in  Ea .
* length: Number of adjacent nodes

**RESULTS**

**MPI:**

Given input:

No of vertices: 3

Adjacency matrix: 111

111

111

The BFS order is: 0->0->1->2

For 9 vertices :

Adjacency matrix:

011100000

000111000

000000100

000000001

000000000

000000000

000000010

000000000

000000000

BFS traversal:

0->1

0->2

0->3

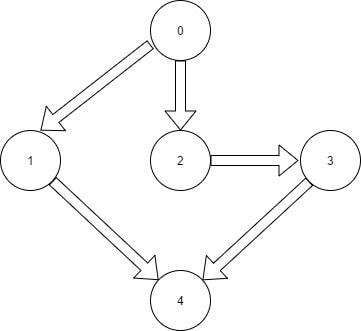
0->1->4

0->1->5

0->6->7

**CUDA:**

**Input:** concatenations of graph shown.



Output:

***For n=5 nodes***

Kernel called 3 times

Order of traversal: 0 1 2 3 4

Step cost: 0 1 1 2 2

***For n=10 nodes***

Kernel called 6 times

Order of traversal: 0 1 2 3 4 5 6 7 9 8

Step cost: 0 1 1 2 2 3 3 4 5 4

***For n=15 nodes***

Kernel called 9 times

Order of traversal: 0 1 2 3 4 5 6 7 9 8 11 10 14 12 13

Step cost: 0 1 1 2 2 3 3 4 5 4 6 5 7 8 6

And so on.

**LIMITATIONS AND IMPROVEMENTS**

The MPI code takes user input and thus there is a substantial time delay in entering the input, which increases program runtime. One possible improvement for this could be reading large input from a predefined input file. The queue structure used has an additional memory overhead.

The CUDA code has a hard coded graph. Although, the graph can be expanded by using loops in the code, graph pattern is rigid. Respecification of start and length values will be required if we want to perform BFS for a graph of a different structure. Only a single block is used which limits the use of only 512 threads for computation.

**CONCLUSION**

The conclusion of the implementation is that massive calculations can be implemented using CUDA and MPI parallel algorithms. However, the programmer’s effort is instrumental in how successful and efficient the algorithm is.